1-1-1980

Simulation of the D-machine.

Lawrence A. Weber

SIMULATION OF THE D-MACHINE

by

Lawrence A. Weber

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

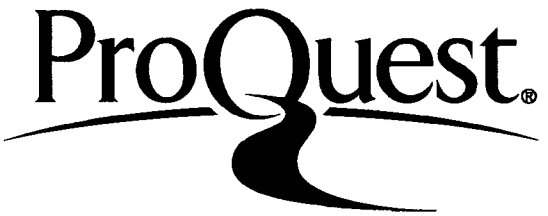
Master of Science

in

Computing Science

Lehigh University

1980



CERTIFICATE OF APPROVAL

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science in Computing Science.

May 8, 1980
(date)

[Samuel L. Gulden]
Professor in charge

[Samuel L. Gulden]
Head of the Division

ACKNOWLEDGEMENTS

The author wishes to thank Prof. S. Gulden for his valuable assistance throughout this project.

TABLE OF CONTENTS

ABSTRACT 1

Chapter 1 INTRODUCTION 2

Chapter 2 PROCEDURE PARSE. 5

Chapter 3 PROCEDURE EXECUTE. 14

Chapter 4 EXECUTION OF DMACH.PAS 20

BIBLIOGRAPHY 26

Appendix A 27

Vita 29

ABSTRACT

This thesis discusses the problems of developing a Pascal implementation of the microprogramming simulator found in 'A Micoprogramming Primer' by H. Katzen.

The problems discussed include parsing a language which contains elements which are LL(5) and simulating concurrent processes in an inherently sequential language.

The result of the work herein described is a fully operational D-Machine simulator on the DEC20. This program is suitable for use in a course on computer organization which discusses microprogramming.

CHAPTER 1

INTRODUCTION

The IBM 370 does not exist. While some people might wish that were actually the case, it is at least in part true. There is not in existence a conglomeration of circuits which behave as an IBM 370. What does exist in numerous Blue Shops are machines which are comprised of circuits which are capable of running programs which simulate the functions of the mythical 370. Those circuits are referred to as the microprogramming level [2], and the programs they run are called microprograms.

This paper is concerned with the microprogramming level of a machine, called the D-Machine, which was used by Burrough's Corporation , and is described in [1]. Specifically, this paper describes the development and use of a Pascal program, which takes a program written in Katzen's reference language, translates the program into microcode, and then executes that microcode as if the microcode were actually on a D-Machine.

The reason that the D-Machine was chosen is that its design is carefully explained by Katzen. With his explanation as a starting point and this program to simulate an actual D-Machine, one can gain experience in microprogramming without fear of disrupting other users. The benefits behind having such experience will become invaluable in the advent of user-microprogrammable computers. It should be stressed that this is an actual machine which is being simulated, and hence presents the programmer with the real problems of machine cycles, non-sequential instruction execution, delays in memory access, and the like. In short, experience with this simulator is experience with actual microprogramming.

There will be little attempt here to recreate a description of the D-Machine given in [1] beyond those points which will elucidate aspects of the translation program. Therefore, before attempting to use this simulator, one should be familiar with [1] or some similar description of the machine being simulated. The following points should give adequate background for understanding DMACH.PAS.

1 - The ALU-input registers are divided into three classes: X-input, Y-input, and Z-input. An ALU operation must contain one X-input and one Y-input. However, a Z-input may function as one or both of those inputs.

2 - An ALU operation is performed during each machine cycle. In the absence of a specified operation '0+0' is performed. However, in actual practice the ALU operation of the previous TYPE1 instruction is performed.

3 - An ALU operation is completed when the next ALU operation is specified by the microprogram. By completion, is meant that the specified destination registers are filled with the results located in the BSW and the ADDEROUT. Those two results are changed however, during each clock cycle to reflect any changes that may have occurred dring the previous machine cycle.

The DMACH.PAS program , which contains approximately 2200 lines of formatted code consists of two level 1 procedures, they are procedure PARSE and procedure EXECUTE. PARSE contains 1500 lines and EXECUTE contains the remainder. These procedures are described in chapters 2 and 3 respectively. Chapter 4 is devoted to instructions and examples on how to use DMACH.PAS, and supercedes Chapter 7 of [1]. While it is required to translate from the reference language to the microcode, there appeared to be no reasons to follow the philosophy of TRANSLANG [1] and make the syntax changes required in [1] for the simulator to function. Hence there are some differences in machine operations between DMACH.PAS and TRANSLANG..

CHAPTER 2

PROCEDURE PARSE

Procedure PARSE might best be described as a non-recursive recursive-descent compiler. As with a standard recursive-descent compiler, such as Pascal compilers, most of the procedure names are linked to the left-hand sides of the reference language BNF description. However, since the BNF description is not recursive neither are the procedures within PARSE.

The target language for PARSE consists of microinstructions which are 16-bits and nanoinstructions which are 54-bits long. With each nanoinstruction there is a corresponding microinstruction by which it is called from the microprogram. While a microinstruction may be a call to a nanoinstruction, it may also be a load immediate instruction into the AMPCR or into either or both the SAR and the LIT registers. Thus, the target language is a mixture of horizontal and vertical architectures.

Procedure PARSE should not be confused with what is normally called an assembler. Even though the machine code produce by the assembler may be on a higher level than the microcode produce by PARSE, the complexity of handling instructions which make use of the horizontal architecture places PARSE on a higher level than most assemblers.

In common with assemblers, however, an instruction may have an optional label. In DMACH.PAS a label may be any length but only the first 10 characters are significant. It must start with a letter, but after that, any combination of letters and digits are acceptable. The maximum number of labels currently allowed is 50 but this can be changed by changing the constant MAXNMFLBLS in PARSE. The number of uses of labels must not exceed the number of allowable microinstructions. But such a situation, would have to be contrived, since each statement can use, at most, two labels, and TYPE 1 statements cannot use any labels at all.

The restriction of the number of label uses to a quantity which would probably never be reached allows one to make PARSE into a one-pass compiler with the label uses being backpatched.

The error recovery of PARSE is somewhat primitive, since it was felt that the complexity of the statements was not so great that one needed to recover from an error within the current statement. If an error is encountered, the user is so notified, and parsing proceeds with the following statement. Even the presence of one error, though will cause the global variable ERRORINPARSING to be set equal to true and hence will cause procedure EXECUTE to not be executed.

For the purposes of the following discussion and for the rest of the paper, we follow the convention of [1] to refer to an instruction which has no associated nanoinstruction as a TYPE 2 instruction, and to refer to an instruction with an associated nanoinstruction as a TYPE 1 instruction. Hence, TYPE 1 instructions are horizontal and TYPE 2 instructions are vertical.

The first problem in parsing and generating code for a language with two distinct types of instructions is to determine the type of the instruction under consideration. Unfortunately, with the reference language [1], being parsed, that is not so easy since it might be necessary to read five symbols of the instruction before it is possible to determine the type of the instruction.

A case in point is the instruction '0 => SAR'. Is it TYPE 1 or TYPE 2? From the preceding paragraphs, you might conclude that since it is loading 0 into the SAR that it is a TYPE 2 instruction. That, by the way is how PARSE would consider the instruction. However, an equally valid interpretation of '0 => SAR' is '0 is the X-input, 0 in the Y-input, the ALU operation is addition, and one of the destination registers (the only one) is SAR.' Hence that way of looking at '0 => SAR' says it is TYPE 1.

If '0 => SAR' were the only problem, there would be no problem, since it could be and has been arbitrarily resolved. However, consider '0 => SAR, X' where X is some string in the language. If the first symbol of X is a digit, 'COMP', or a statement label, then the instruction is TYPE 2 else the statement is TYPE 1. Hence '0 => SAR, 4 => LIT' is TYPE 2, and '0 => SAR, LCTR' is TYPE 1.

There are similar problems with '1=>SAR', '0=>AMPCR', and '1=>AMPCR'.

In its final form PARSE resolves the difficulty of determining the type of an instruction while generating code, by not generating any code until the type of the instruction is determined. This is justified by the complexity of the resolution and has only one drawback. This drawback is that the instructions must be buffered. This is not serious , though, since even the longest

instruction in the reference language can fit onto an 80 character line.

After the type of the instruction has been determined, the current character pointer is moved to just after the label if the statement is labelled, else it is set to 0. At this point, control is passed to either GENTYP1 or to GENTYP2, depending on the outcome of the determination of type. It should be noted that the determination process is biased towards TYPE 2, hence, an error in typing a TYPE 2 statement might make the determination say that the statement is TYPE 1, and an error could be returned from GENTYP1 when it discovers that the statement is not in fact of TYPE 1.

GENTYP2 is by far the simpler of the two procedure and operates only on 16-bit instructions. The only problem is that if either the SAR or LIT is being filled then the AMPCR cannot be filled simultaneously. Also, Two different values cannot be placed into the same register in the same instruction.

GENTYP1 places a F000 HEX into a 16-bit microinstruction and then fills in the low order bits of that instruction with the binary representation of the next address which is available in the nano-memory.

The operation of GENTYP1 is straightforward with the exception of the determination of the adder operator. Other than that it checks to see if ALU or MDOP-CAJ operations are conditional or unconditional following the rules of the language, and sets the SC=0 and SC=1 successor. The default successor is STEP.

As just mentioned the sole problem encountered in GENTYP1 is the determination of the adder operator. This problem is caused by the reference language in a effort to make the writing of microprograms easier.

The general form of the logical operation specification is <not><Xselect><operator><not Yselect>. A problem arises in that the language allows X and Y selects preceded by 'not' but, the D-Machine has no such facilities for handling X or Y inputs preceded by a 'not'. It can only operate on an X-input and a Y-input with one of its 16 defined operations. It is therefore difficult for the translator to determine if a given syntactically correct operation is possible and if so to generate the appropriate adder operation.

The method that GENTYP1 uses to determine the translations of one operation which contains 'not' 's into an operation which contains no 'not' is through the use of sets which have as their base set a scalar type called ADDOPTYPE. These sets are filled with up to four items. 1. whether X or NOTX is used, 2. what operation was specified, 3. whether Y or NOTY was used, and 4. whether the number 1 was added to or subtracted from the Y-input.

After the parsing of the adderoperation has been completed and the set has been filled with the appropriate items, procedure SELECTADDEROP compares the resulting set against sets for which it knows the appropriate adder operation. If neither NOTX or NOTY is in the set then the appropriate adder operation is the one which was specified by the microprogram. If however, the set is [NOTX, OROP, NOTY] then the generated adder operation would be NAN. If the set is [NOTX, ANDOP, NOTY] then the adder operation would be NOR. If the set is [NOTX, ANDOP, Y] then the adder operation generated would be NRI, and so on. If the compiler recognizes that there is no corresponding operation for that which is indicated in the microprogram, an error condition exists, and the programmer is so notified.

Finally, a word about how PARSE handles program tokens. Since there are a total of 71 reserved words, and 10 symbols all of which can act as symbols, it was not possible to use a single SCALAR type as is done in Pascal and PL0 [3]. The reason is that on the DEC20 a scalar type can have a length of at most 72 identifiers, and the CDC is limited to 59 identifiers. It was decided that three scalar types would be used, SYMTYPE, SYM1TYPE, and SYM2TYPE. SYMTYPE holds those identifiers that are associated with symbols, while SYM1TYPE holds those identifiers which are asscoiated with the alphabetically first 36 reserved words, and SYM2TYPE holds those identifiers associated with the final 35 reserved words.

While on the DEC20, SYM1TYPE and SYM2TYPE could be easily combined, DMACH.PAS was written to be transportable to all Pascal's currently in use at Lehigh, and hence a length of 71 in a scalar type would be unacceptable.

Using three scalar types for the purposes of identifying tokens does provide some difficulties, however. One must know which of SYM, SYM1, or SYM2 holds the last token value. This was resolved by introducing the null-states NUL, NUL1, and NUL2 for SYMTYPE, SYM1TYPE, and SYM2TYPE respectively. At any point in the program the procedure GETSYM makes sure that only one of SYM, SYM1, and SYM2 is not in its null-state. The one which is not is the value of the last token, produced by GETSYM from information

received from the input stream.

CHAPTER 3

PROCEDURE EXECUTE

Whereas procedure PARSE was a reasonably straightforward, though tedious, exercise in language translation, procedure EXECUTE had to include features of a microprogramming level with which the writer had had no experience prior to the start of this project. In a conventional higher language programming system, one expects the order of statement execution to be sequential, when one reads from memory one expects to be able to use the value in the next statement, and when one writes to memory, he expects that he can do so again in the next statement.

While some of those difficulties are also encountered in assembly programming, there is also the problem of simultaneity. Processes which might affect each other are travelling along different paths concurrently. Since it is obviously impossible to create simultaneous processes in Pascal, the processes had to be analyzed to determine which sequence of sequential execution of the processes would

simulate simultaneous execution of the processes.

A simpler though related cause for extra programming is that when using Pascal, one is unaware of the delay in memory access. One must therefore, artificially slow down the process of memory access through the use of temporary holding variables, which at the appropriate time will be sent to their designated destinations.

However, by far the most difficult aspect of EXECUTE is the handling of the machine cycles. This should be expected since the handling of machine cycles is considered to be the most difficult part of microprogramming in general.

Since the problem of memory reads and writes is the simplest, it will be discussed first. At the time of a memory write call, the appropriate register is accessed for the address of the write location. This address, which is in binary, is converted to decimal and assigned to the variable WRITEADDRESS. The contents of the MIR is the copied into a holding register called the WRITEBUFFER. Next the SAICLOCK is set equal to the current value of CLOCK + 2. At CLOCK = SAICLOCK, SAI will be assigned the value true, and the contents of the WRITEBUFFER will be copied into SMEM[WRITEADDRESS]. Note that the initiation of a memory write does not set SAI to false. SAI is one of the static

conditions, each of which is set equal to false by testing its value.

Memory reads are similarly programmed. At the initiation of a memory read, the value of the appropriate register is assigned to the variable READADDRESS, and RDCCLOCK is set to CLOCK + 2. When CLOCK = RDCLOCK, SMEM[READADDRESS] is assigned to the EXTERNAL register, and RDC is set equal to true. As with SAI, initiation of a read does not set RDC to false.

A question might arise as to why it was decided that 2 clock cycles would be sufficient for a read or a write, since Katzen makes no mention of how large a delay is involved. A two cycle delay was chosen to make sure that the beginning microprogrammer would not be able to use the value in the next machine cycle, since on the actual machine the read would not yet be complete. However, more than 2 cycles would simply increase the number of cycles required for execution and would yield no beneficial results along the lines of experience with microprogramming. However, at the intial run of the simulator, it was discovered, that Katzen's simulator also waits 2 cycles. While the coincidence was somewhat surprising, it seems to substantiate the previous reasoning.

The problem of cycles and phases will now be examined. During each machine cycle, the microinstruction which is located at the address indicated by the variable NEXTADDR, is fetched and is said to be in Phase 1 of its execution. This is the same for both Type 1 and Type 2 statements. Also during each machine cycle, the pending ALU operation is initiated, but not necessarily completed. The pending ALU operation is either, the last specified operation from a Type 1 instruction, or is '0+0=>' if there have either been no previous Type 1 statements in the program, or if the previous Type 1 statement did not specify an ALU operation. In any case, some operation is performed and on the basis of its result, the dynamic conditions are changed.

If the instruction which is going through its Phase 1 is of Type 2, then its execution will be completed during the current machine cycle. NEXTADDR will become NEXTADDR+1, and if there is a Type 1 instruction in a Phase 3, its phase is converted into a phase 2 holding phase.

If the instruction which is going through its Phase 1 is of Type 1, then the events are more complicated. If the instruction contains a conditional, then the results of the current static conditions, and those of the dynamic conditions set during the current cycle may be used.

The following cases are next checked: 1) Is the specified ALU operation under the control of a condition and is the condition true, and 2) Is the ALU not under the control of a condition. If either case holds, the Phase 3 of the previous Type 1 instruction is completed by sending the results of the ALU operation completed at the beginning of this machine cycle to the appropriate destination registers. If neither case holds, the previous Type 1 instruction which is in Phase 3 is moved to a holding Phase 2, and the current instruction has no Phase 3. This was denoted in DMACH.PAS by using a BOOLEAN valued variable called PHASE1HASPHASE3.

At this point, if a Phase 3 was just completed, the 36-bit control register is filled with those bits from the Nano-memory which will be necessary to specify the ALU operation to be be executed during the start of the next machine cycle.

After the ALU operation is processed, the following two cases are checked : 1) Are the MDOP-CAJ operations under the control of a condition and is that condition true, and 2) Are the MDOP-CAJ operations not under the control of a condition. If either case holds then, the specified operations are performed. It should be noted that if the specification of an ALU operation in the current Phase 1 instruction caused the Phase 3 of the previous Type 1 instruction to be completed that those results may be used

for the current MDOP-CAJ operations. For instance, suppose that the previous Type 1 instruction caused the MIR to take a new value. It is this new value and not the previous value which will be written to memory by a memory write instruction in the current Phase 1 instruction.

CHAPTER 4

EXECUTION OF DMACH.PAS

A most important point to know about any programming language translator is the form and location, at which the translator expects to find the program that is to be translated from the host language into the target language.

The form that DMACH.PAS expects is free form with the inevitable few exceptions, they are: 1) There can only be one statement per line, and every statement must be contained on one line; 2) Spaces and tabs may be used freely except between a label being declared and its subsequent colon, or in the middle of an identifier.

Several differences between the symbols in Katzen's TRANSLANG [1] and DMACH.PAS must also be noted. However, the differences exist in order that DMACH.PAS might resemble the reference language more closely. In fact the only difference between DMACH.PAS and the reference language is that the rightward-ponting arrow had to be replaced by the

two adjacent symbols '=>', since there is no corresponding character on the system. Other than that a colon is used for a colon, a percent is used for a percent, and operators need not be surrounded by spaces, all of which are the case in TRANSLANG. Also, for ease of reading, blank lines may be freely inserted into the microprogram.

In the example programs found in Appendix A, the presented style of starting labels in the first column, then tabbing to start the statement body is recommended for clarity, but is not required by the translator. A last point regarding differences between DMACH.PAS and TRANSLANG is that DMACH,PAS does not require that a '%' end each statement, but only if if one wishes to place a comment at the end of the line following a statement.

If this program is to be moved to the CDC Pascal3, since the local system uses the CDC 63 Character Set. The '%' should be changed to a '$', and the reference to tab in procedure GETNONBLANK must be removed.

On the subject of where DMACH.PAS expects to find the microprogram, it must be the first data in the file typed in response to the 'INPUT :' prompt of te TOPS20 oprating system after the command 'EXECUTE DMACH.PAS' has been issued.

Before the program is parsed and the code is generated, the user is asked if he wishes to see the binary code which is created by PARSE. If the user does so wish he should respond 'N' to the prompt 'SUPPRESS BIT PATTERNS?(Y/N)'. The latter response will cause code to be output following the printing of the microprogram text. However, it is not necessary to see the code for the code to be executed. Also, since on a CRT, one is limited to a 72 character field, if a micro-instruction refers to a nano-instruction, only the address of the nano-instruction and the nano-instruction itself are output, and not the micro-instruction. No information is lost, since the presence of an address and a nano-instruction implies the existence of the 'missing' micro-instruction.

There is no facility in DMACH.PAS for outputting the HEX of the microcode. For, while the bit format of the microcode is hard to follow, the HEX format is indecipherable. At this point, if errors were found in the microprogram, DMACH.PAS will cease execution. If not, then the user is asked if he wishes to procede with the execution of the microcode which PARSE has produced. If he so desires, he should respond 'Y' to the prompt 'EXECUTE?(Y/N)'. Before execution will occur, however, the following questions must be answered:

'INPUT DATA IS BINARY(B), OCTAL(O), DECIMAL(D)' - Depending on whether a B, O, or D is input, any data

read from the input file, or input into the program is interpretted as being of the corresponding base.

'OUTPUT DATA IS BINARY(B), OCTAL(O), or DECIMAL(D)' - Same as above except that this refers to the memory dumps, and output point register dumps. It need not be the same as the previous type.

'LOAD SMEM FROM INPUT FILE?(Y/N)' - A 'Y' will indicate that in the input file following the microprogram, there is data which you wish loaded into the S-memory starting at location 0. This could be an S-memory program or simply data to be used by the microprogram.

'STARTING ADDRESS = ' - Indicates the address in micro-memory at which the user wishes to start execution. Saves clock cycles if the beginning of a micrprogram consists of a directory.

'Maximum Number of Clocks to Simulate = ' - Sets a limit on the number of cycles so that an endless loop will not result.

'Begin Output Point Dumps at MPM address = ' - Point at which to start displaying contents of ALU registers, and conditions.

'End Output Pont Dumps at MPM address = ' - Point at which to start displaying contents of ALU registers, and conditions.

'Number of Clocks between Output Points' - Between the two addresses above there, will be the number of cycles that is here specified between dumps of information.

'S-memory Dump after Execution?(Y/N)' - Allows the display of selected registers in the S-memory after execution to determine the effects of the microprogram. A 'Y' will alow a dump.

'Enter S-memory in Consecutive Blocks'

'Enter 9999 for Starting Address When Finished' - You will be continaully prompted for a starting address, a finishing address, and for the contents to be placed in those addresses, until you enter 9999 for a start address. The start address being greater than the finish address, or the finish address being greater than MAXSMEMLOC will generate error conditions and terminate the program.

At this point the execution will procede until the END statement of the program is reached, the number of MAXCLOCKS is reached, or an error condition is generated.

In all cases DMACH.PAS will print 'END OF SIMULATION - REGISTERS CONTAIN', and perform an output point register and condition jump.

If in the above, the user requested a memory dump, then he will see displayed 'Memory Dump requested', 'Enter 9999 as starting address when done'. He will then be prompted for starting and finishing addresses for S-memory dumps. The memory will then be dumped to the terminal. This process is repeated until 9999 is input in response to a request for a starting memory address.

BIBLIOGRAPHY

1. Katzen, H. A Microprogramming Primer McGraw-Hill, New York, NY, 1977.

2. Tannenbaum, A. Structured Computer Organization Prentice-Hall, Englewood Cliffs, NJ, 1976.

3. Wirth, N. Algorithms + Data Structures = Programs Prentice-Hall, Englewood Cliffs, NJ, 1976.

APPENDIX A

The program execution on the following page is the result of the execution of DMACH.PAS. The program which is being run places 1's into all bit positions from S-Memory addresses 1 through 10.

A copy of the source of DMACH.PAS is on file at the office of Computing and Information Science, Department of Mathematics, Lehigh University.

```
 EX DMACH.PAS
LINK:   Loading
[LNKXCT DMACH Execution]
INPUT       : 42
SUPPRESS BIT PATTERNS?(Y/N) N
0000    0 => A1, LCTR
0001    9 => LIT
0002    RPT -1 =>AMPCR
0003    NOT 0 =>MIR, INC
0004    RPT: A1 + 1 =>A1, MAR1
0005    MW1, IF SAI
0006    WHEN SAI THEN STEP
0007    IF NOT COV THEN INC, JUMP ELSE STEP
0008    END
MICROCODE FOLLOWS

0000     0 0000 000 000 001001 000 0000000 0 1110 00 100 0000 0 0 000001 00 0000
0001   1110000000001001
0002   1100000000000011
0003     1 0000 000 000 001001 000 0000000 0 1000 00 000 0000 1 0 000010 00 0000
0004     2 0000 000 000 001001 101 0000011 0 0000 00 100 0000 0 0 101100 00 0000
0005     3 1001 000 000 001001 000 0000000 0 0000 00 000 0000 0 0 000000 00 0110
0006     4 1001 000 000 001000 000 0000000 0 0000 00 000 0000 0 0 000000 00 0000
0007     5 1000 110 000 100001 000 0000000 0 0000 00 000 0000 0 0 000010 00 0000
0008   0100000000000000
EXECUTE?(Y/N) Y
INPUT DATA IS BINARY(B), OCTAL(O), DECIMAL(D) - O
OUTPUT DATA IS BINARY(B), OCTAL(O), DECIMAL(D) - O
START OUTPUT AT MPM ADDRESS - 99
END OUTPUT AT MPM ADDRESS - 99
S-MEMORY DUMP AFTER PROGRAM TERMINATION?(Y/N) Y
MAXIMUM NUMBER OF CLOCKS TO SIMULATE - 100

ENTER S-MEMORY IN CONSECUTIVE BLOCKS
ENTER 9999 FOR STARTING ADDRESS WHEN FINISHED - 9999

END OF SIMULATION - CURRENT STATE IS

AMPCR=            3
P1=8 P2=6 P3=-1
A1=10 A2=0 A3=0 B=0 ADDOUT=0
CLOCK =           55
LIT=9 CTR=255
LC1=false LC2=false LC3=false RDC=true SAI=false
MST=false LST=false ABT= false AOV=false COV=false

MEMORY DUMP REQUESTED

INPUT START S MEMORY ADDRESS   1
INPUT FINAL S MEMORY ADDRESS   15

37777777777   37777777777   37777777777   37777777777   37777777777
37777777777   37777777777   37777777777   37777777777   37777777777
00000000000   00000000000   00000000000   00000000000   00000000000
```

VITA

Lawrence Adam Weber was born on July 2, 1956 in Philadelphia, Pennsylvania. He is the son of Dr. and Mrs. George Weber of Broomall, Pennsylvania.

In 1974 he entered Dickinson College in Carlisle, Pennsylvania, from which he received a Bachelor of Science in Mathematics in 1977. In fall of that year he entered Lehigh University in Bethlehem, Pennsylvania from which he received a Master of Science in Mathematics in 1979. At the present time he is working as a teaching assistant for the Department of Mathematics, Lehigh University.